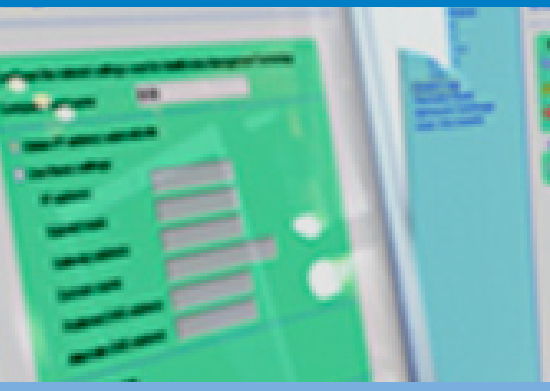
Volume 11 Issue 04 Published, November 15, 2007 ISSN 1535-864X DOI: 10.1535/itj.1104.06



Intel® Technology Journal

Multi-Core Software

Methodology, Tools, and Techniques to Parallelize Large-Scale Applications: A Case Study

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Methodology, Tools, and Techniques to Parallelize Large-Scale Applications: A Case Study

Knud J. Kirkegaard, Software and Solutions Group, Intel Corporation
Mohammad R. Haghighat, Software and Solutions Group, Intel Corporation
Ravi Narayanaswamy, Software and Solutions Group, Intel Corporation
Bhanu Shankar, Software and Solutions Group, Intel Corporation
Neil Faiman, Software and Solutions Group, Intel Corporation
David C. Sehr, Software and Solutions Group, Intel Corporation

Index words: multi-core, parallelism, threading, data dependence, privatization

ABSTRACT

Multi-core processors are now mainstream, while many-core architectures are arriving. Yet getting general-purpose software ready to take full advantage of the available hardware parallelism remains a challenge. There are, in fact, very few success stories of semi-automatic parallelization of large-scale integer applications outside the high-performance computing (HPC) and transaction processing domains. In this paper, we report on such a success story: threading the Intel® C++ Compiler [3] which resulted in an average 2x speedup in compiling a range of CPU2000 benchmarks. We present the methodology and tools that enabled us to achieve this success. We believe our approach is generally applicable to threading a large class of applications.

INTRODUCTION

In this paper we focus primarily on the techniques used to parallelize an application, the tools that facilitate the parallelization, and the new insights this approach yielded. Techniques that proved helpful in our work are at the core of a comprehensive solution suite Intel is developing to assist software developers discover and exploit parallelism in their applications. The generally applicable source changes necessary to make the compiler thread safe are also categorized and described. As expected, good software engineering principles such as modularization, data abstraction, and information hiding ease the process of threading an application. We also describe how we automated repetitive source changes. To make it feasible to apply all the source changes necessary for an application of this size, where the threaded loop spans hundreds of modules covering hundreds of thousands of lines of code with extensive use of macros, semi-automated script tools were developed. It is easy to get overwhelmed by the data dependence complexity and size when starting to thread existing serial applications, but as we hope to illustrate in this case study, with the help of Intel's threading tools and a systematic approach, it is possible to achieve large application threading with a reasonable amount of effort and time.

The threading effort, involving a small team over a relatively short period of time, successfully yielded a working parallelized compiler. Although work remains to be done in tuning the resulting application, we also discuss in this paper the impact of different thread scheduling algorithms and the speedups achieved. We also briefly discuss the issues involved in maintaining a thread-safe application.

DESCRIPTION OF THE APPLICATION

The Intel Compiler is a large non-numeric application that compiles C/C++ and Fortran applications for a variety of Intel® platforms including the IA-32 architecture, the Intel® 64 Architecture, and the Itanium® processor. Despite having evolved over the years to target new Intel® processors and platforms, parallelization of the compiler itself was not an initial design goal. As such, the compiler has characteristics similar to other large integer applications that need to be parallelized in order to take full advantage of multi-core platforms. At the Intel Compiler Lab, we parallelized the Intel Compiler and achieved great performance results. One of our goals was to fully understand the issues that application developers encounter when parallelizing a large-scale application.

We chose to thread the Intel C++ Compiler for a number of reasons.

First, we had detailed knowledge about the application. We strongly believe that to thread an application successfully, it is important to involve the application architects as they tend to know what to parallelize and what not to parallelize. Moreover, an in-depth knowledge of the application global data is crucial.

Second, the compiler has evolved over the years and therefore is a good proxy for real-world, legacy product applications. It is a mature integer application that was not initially designed to be thread safe.

Third, by choosing a non-numeric application outside the traditional high-performance computing (HPC) domain, we strived to address the challenges other application developers would encounter when undertaking a similar task. A particularly interesting challenge is that the potential parallel region spans hundreds of source modules containing millions of lines of code. In contrast, in typical HPC applications, the parallel loops are contained within one module or even just one function.

Finally, there is an inherent scalable parallelism in what compilers do. By using performance analysis tools and built-in timers in the application itself, we found that the region we intended to parallelize accounts for up to 80% of the application time in compiling a number of benchmarks. With infinite parallelism there is a theoretical speedup of 5x as dictated by Amdahl's law. If S is the fraction of the program that is serial and N is the number of available processors, the speedup through parallelism is $1/(S + ((1-S)/N))$, and the theoretical speedup limit is $1/S$. For example, if 80% of the application time is in the parallel region, then S equals 0.20, and assuming $N \rightarrow \infty$, we get at best a 5x speedup through threading.

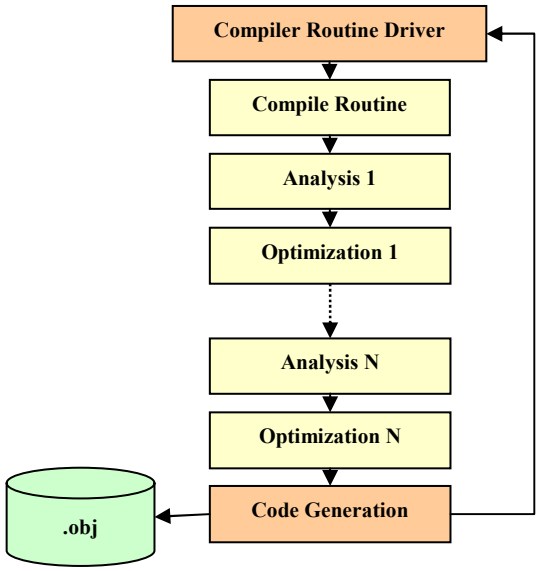


Figure 1: Serial execution of the compiler driver loop

The basic flow of the compiler is shown in Figure 1. After the front-end parses the input program into an intermediate representation, the compiler iterates over the functions of each module. At each iteration, the compiler translates the code of the corresponding function, applies a series of optimizations to the intermediate representation, and finally generates code for the function. We observe that each routine compilation is logically independent of each other; that is, we can change the order in which routines are compiled without affecting the correctness of the program and therefore it is legal to parallelize the loop that compiles each individual routine. This loop spans almost 200 source modules containing roughly half a million lines of mostly C source code. The flow of the parallelized compiler is shown in Figure 2.

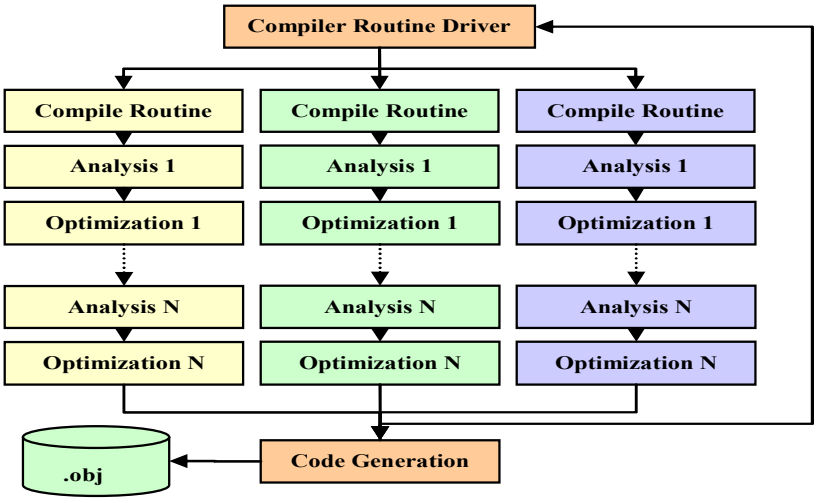


Figure 2: Parallel execution of the compiler driver loop

Table 1: The execution profile of the compiler across its loop hierarchy

%Ticks	Ticks	Entry	Exit	File:Line:Col	Function:Line
51.7	675952724	1	1	ip/placement.c:657:39	compiler_driver:276
18.0	235966016	14782	14782	fe/lexical.c:9589:8	get_token:9509
16.1	211070764	1	1	fe/decls.c:14518:7	translation_unit:14480
6.5	84369904	1	1	intrin/intrin.c:1536:5	intrin_process:1517
4.7	61261408	2	2	fe/code.c:3474:3	dump_routines:3423
4.5	59420636	87	87	il/verify.c:3014:5	verify:3011
4.2	54363924	217	217	fe/preproc.c:460:5	skip_endif:442
4.1	54079124	4228	4228	fe/lexical.c:6706:8	skip_space:6661
3.9	51635388	27	27	fe/lexical.c:4449:5	search_input:4416

Of course, we could have parallelized the compiler at a higher or a lower level. The highest level would simply be to compile the modules of an application in parallel, as with a parallel make file. This scheme is very simple to implement. However, it can easily run into load-balancing problems when the application's modules have widely varying sizes. It also fails when the build uses "link time compilation," an important feature of our compiler. Link time compilation pre-compiles the individual modules of the application into intermediate representations and then processes all the intermediate representations at once in a single execution of the compiler, making it possible to obtain the benefits of inter-procedural optimizations across the entire application.

At a lower level, we could have looked at smaller potential parallel regions, such as individual optimization phases. It might be easier to parallelize these phases than to parallelize the entire compiler driver loop, but any one piece would have accounted for only a small fraction of the total compilation time. Therefore, it would have been necessary to parallelize many smaller pieces to get any significant benefit from threading. Furthermore, working with the outermost driver loop allowed us to learn more about the problems of threading very large applications.

THE THREADING METHODOLOGY

We followed a threading methodology that consists of the following four basic steps:

1. Discovering parallelism
2. Expressing parallelism
3. Debugging the threaded code
4. Tuning the threaded code

In the first step, the application architect needs to discover the parallelism that is available in the application. Tools that provide loop-profiling capabilities can be used. One would need to know the execution profile of the application across its loops. This includes both the loops in the program control-flow graph as well as the loops in its call graph. In our case, a significant majority of the execution time of the compiler, as explained before, is spent in the body of the compiler driver loop. As contributing architects of the compiler, we knew where that loop was, but we found a loop profiling capability of the compiler generally helpful for threading. Through an option, the compiler instruments the generated binaries with timing instructions before and after program loops and functions. The execution time profile of the compiler across its loop graph is shown in Table 1. This option may also be provided through dynamic instrumentation tools such as the Intel VTune™ Performance Analyzer [5]. The application architect can go through the application loops in a top-down fashion ordered by the total contribution of the loop to the execution of the application. If, intuitively, the loop has parallelism potential, then the architect would need to know how many data dependence violations would be violated should that loop be parallelized.

Data Dependence

The notion of *data dependence* captures the most important properties of a program for parallel execution at all levels [1, 6]. At the loop level, the dependence relation is defined in three categories as follows.

1. If an iteration of a loop writes to a memory location that is later read in another iteration of the loop, we say that the second iteration is *flow-dependent* on the first iteration.

S1: x = …
S2: … = x

2. If the first iteration reads from a location that is later modified in another iteration of the loop, we say that the second iteration is *anti-dependent* on the first iteration.

   S1: … = x
   S2: x = …

3. Two iterations of a loop are *output-dependent* on each other if both write to the same memory location.

   S1: x = …
   S2: x = …

Data dependence relations are often called *hazards* or *data races*. Flow dependence, anti-dependence, and output dependence relations are equivalent to Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW), respectively.

A loop that contains no dependence relations can be parallelized. On the other hand, parallelizing a loop that contains any of these dependence relations may cause invalid results. However, it can be shown that if a loop contains only anti- and output-dependence relations, it can be parallelized with the proper code change [1].

Therefore, in order to parallelize an application, the application architect needs a tool to identify the dependence relations between its possible threads of execution such as various iterations of its loops. In our threading experience, we used the Intel® Thread Checker [2, 4], a software tool that helps developers detect the race conditions [7, 8] in their threaded applications. Among its many features, Thread Checker has a mode of operation, called *projection mode*, which is particularly helpful for parallelization. In this mode, the user can mark a sequential loop as a parallel loop. Thread Checker will run the code sequentially, but with some additional bookkeeping to reveal the race conditions that would occur should that loop actually run in parallel. This mode is extremely helpful in parallelization as it allows the sequential application to run to completion while the information about its possible threaded execution is being collected. More specifically, in spite of the data dependence violations in the parallel execution of the application, Thread Checker's projection mode does not crash due to such violations. We marked the compiler driver loop as a parallel loop and ran it under the control of Thread Checker on a small test program that included a single file with a few functions, conditional statements, and loops.

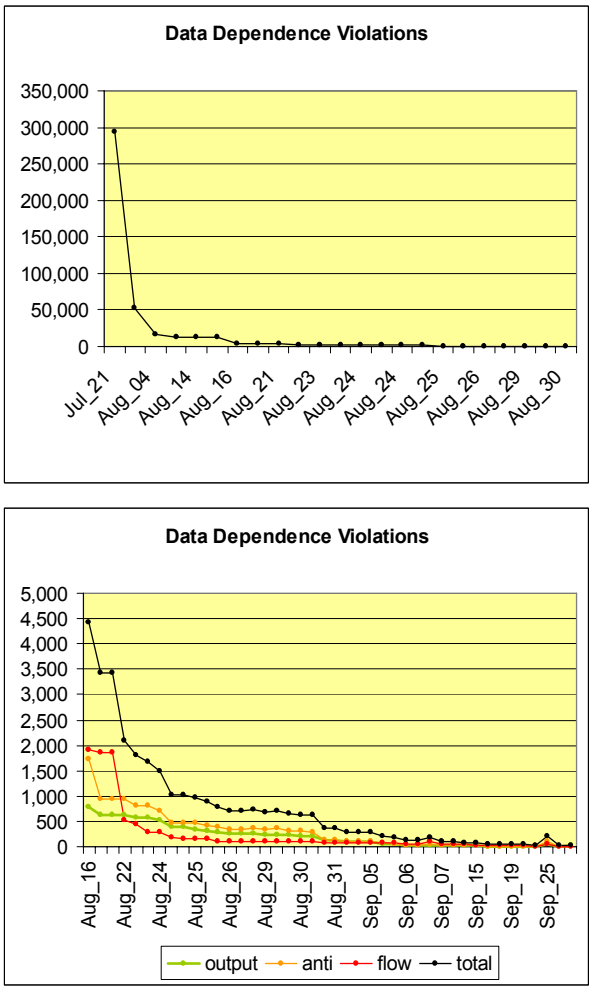


Figure 3: Progress of elimination of dependence violations over time

We also used the Intel Compiler code-coverage tool to make sure that our simple test resulted in reasonably good coverage of the compiler source code. In particular, we made sure that most of the critical components of the compiler including its various optimizations were exercised when compiling our test program. One should note that dynamic analysis tools, such as Thread Checker, typically provide information only about what occurs in a particular instance of program execution as opposed to static tools that may be able to provide information about what can possibly happen in the program execution in the general case. Thus, the lack of dynamic dependence violations does not necessarily imply thread correctness. The use of the code-coverage tool alleviates this problem to some extent. If one does not observe any dependence violation in a piece of code, and the coverage information reveals that the code was not in fact executed, then nothing can be inferred about the possible dependence relations in that piece of code. The first run of our

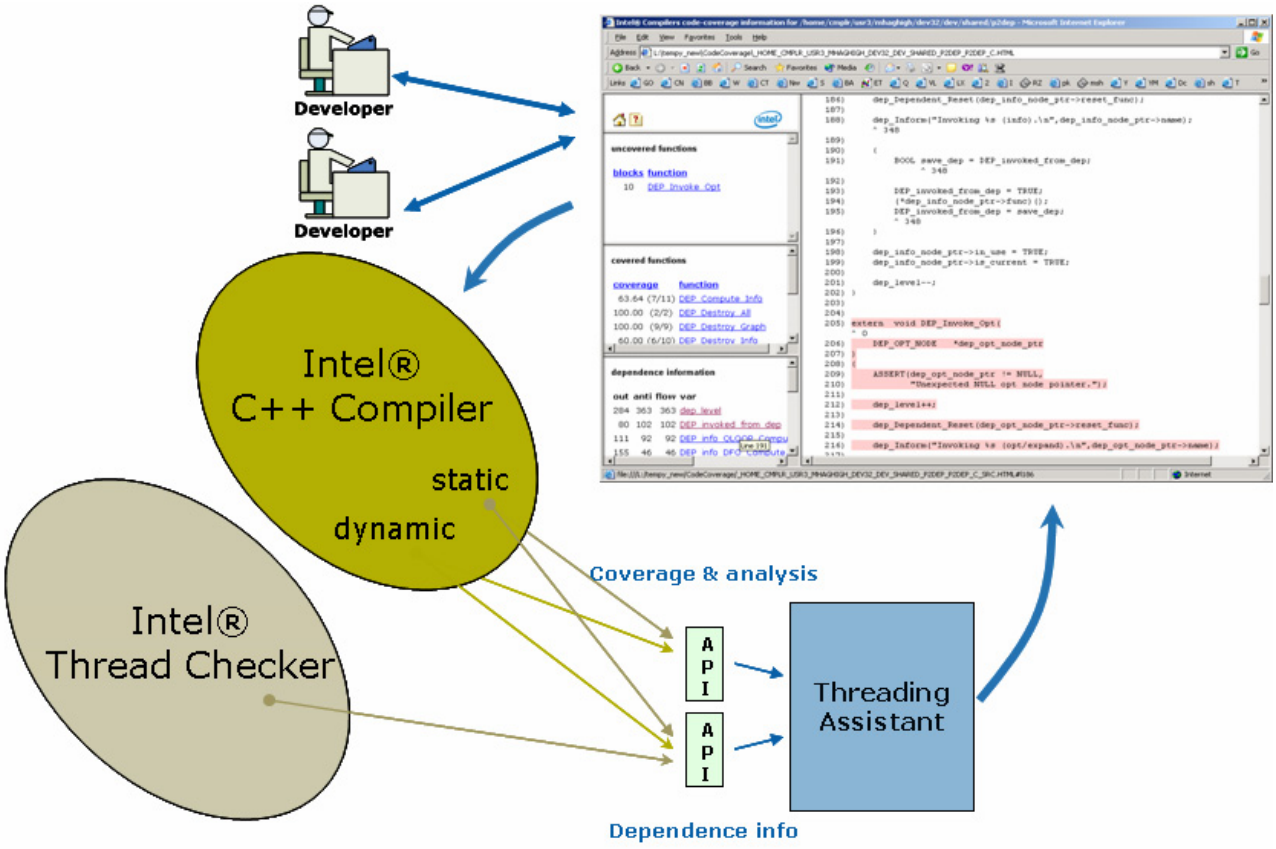


Figure 4: Flow of the iterative process of dependence-violation elimination

instrumented compiler under Thread Checker took several hours to complete and resulted in about 300,000 data-dependence violations. Such an error size is well above the comfort zone of most of the available race-detection tools.

Managing the Size Problem

The key to managing the large dependence problem size is controlling the precision of the generated analysis data. In the early phases of the threading effort, one may not need all the details about every individual dependence violation that is detected, including information about the source position of the two memory accesses that are involved and the call stack of each of them. At a later point, however, such information may actually be crucial to figure out the exact conditions under which the violation occurs.

Thread Checker already supported several useful filtering capabilities, such as filtering based on file names, variable names, and so on. It also summarizes the violations that have identical first memory access source position and base, and those that have identical second memory access source position and base. This filter effectively groups the violations that occur when processing the data in a given array in a single loop with the same dependence distance. In addition to the existing filters that Thread Checker supports, we developed a new filter that proved very effective at grouping the violations that map to different source files and functions and thus reduced the problem size dramatically. In this filter, we grouped together the violations whose base addresses were identical, irrespective of their source file positions and functions. One can think of this heuristic as projecting the dependence information based on its data structure as opposed to based on the code. We then picked the source position of the first such violation as the representative of that group of violations and summed all the violations in that group. Using this technique, we immediately realized that approximately 65% of the violations corresponded to the compiler memory pool data structure. What we lost in this filter is all the details about every individual violation, but what we learned was sufficient to guide us to make the pool thread safe and eliminate almost 200,000 dependence violations with a small number of changes to the source code. After fixing this problem, the subsequent instrumented runs not only have a much smaller problem size but also a much shorter turnaround time. The reason for this is that the runtime overhead of race detection depends on the number of violations, and by eliminating the violations in a prioritized fashion, we constantly speedup the process of the next iteration. Figure 3 shows the number of dependence violations over time.

The interactive development environment we created to assist us in the parallelization effort is illustrated in Figure 4. The main components of this platform are Intel Thread Checker, Intel Compiler, and Intel Compiler's code-coverage tool. In this framework the dynamic dependence diagnostics produced by the Intel Thread Checker and the dynamic code-coverage information generated by the instrumented binaries are combined with the static information provided by the Intel compiler to collectively assist the parallelization effort. The communication of information between these components is facilitated by means of well-defined APIs. The collected information is then assimilated by our Threading-Assistant analyzer to produce a compact set of dependence violation diagnostics and threading hints to the developers. The parallelization process is iterative and may require several iterations before all the dependence violations are eliminated and thread-safe code is obtained.

Making the Application Thread Safe

After identifying the loop to be parallelized in the *Discover* step of the threading methodology the application must be made thread safe with respect to that loop. Identifying all the global data with dependence relations and effectively privatizing them was by far the largest part of our effort and is a challenge for an application of this type. We spent about 10 person months to achieve thread safety for the compiler at the optimization levels chosen for our prototype project. In order to achieve this goal in the given project time frame, we not only relied on threading tools but also developed scripting tools to assist us in applying the needed source-code changes semi-automatically. Looking at the global data dependence violations and knowing the modular structure of the compiler, we found it useful to categorize statically allocated global data, as opposed to heap allocated global data, into three categories. For each category of global data we have a method for making the data thread safe:

Global Data with Dependence Relations

Initially, we attempt to rewrite the code in these data to eliminate the data dependence, and if that is not possible we have to apply locks to synchronize the access to the global data.

Global Data Defined Outside the Loop

This category includes global data that are defined outside the parallel loop and only read inside the loop. This is a thread-safe usage of global data and doesn't require any rewrite. The main issue is to ensure that the usage of the global data remains thread safe.

Global Data with Restricted Scope

This category consists of global data that could have been declared as constant or as stack variables. If it is possible to rewrite global data to be constant or as stack variables they become thread safe automatically. This, furthermore, improves the software engineering aspect of the application.

In the first category, where we have flow-dependence relations, we found there were many false flow-dependence relations that can be eliminated by privatizing the data and thereby improving the software engineering. One example is global data shared across loop iterations where the data need to be reset to proper initial values for each iteration of the loop. We found it useful to categorize the global data with data dependence relations into four sub categories:

1. Synchronized
2. Mutable
3. Persistent
4. Transient

The synchronized category includes the global data that require locks for controlled synchronized accesses. It is not possible to privatize such data without extensive changes. Examples of global data that require synchronization are input/output operation and heap allocation management.

The mutable category contains the global data that generally are defined before the parallelized loop, but they may be modified by an iteration of the loop (only to be reset to the original value before the next iteration). Mutable data are privatized by creating a thread-private copy of the data for each of the iterations. This has the additional advantage that there is no longer a need to restore values for the next loop iteration, if data were modified. Furthermore, this helps improve maintainability of the code by eliminating the code necessary to restore values.

The persistent category comprises the global data that are defined and used in each thread but do not have any cross iteration dependence relations. In the case of the compiler, examples of data in the persistent category include the intermediate representations for routine statements, expressions, symbol tables, control flow graphs, etc. The lifetime of the global data in the persistent category spans the entire thread. They are allocated and initialized after thread creation and freed before thread termination. Allocated persistent data are assigned and accessed through a thread-private pointer. In object-oriented terms, the state object is constructed after thread creation and destroyed before thread termination.

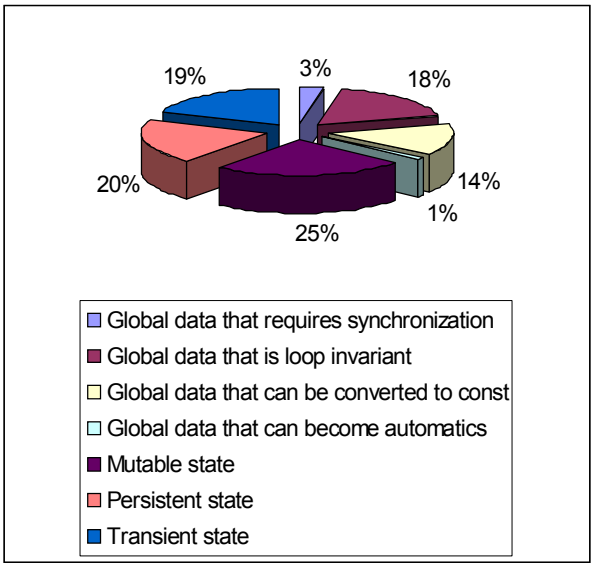


Figure 5: Breakdown of the global data

The transient category consists of the global data that are defined and used only within a certain phase of the threaded region, for example global data that are used to do constant propagation. Global data in the transient category are allocated on entry to a phase and freed on exit from that phase. In general, the transient state is allocated on the stack and is assigned and accessed through a thread-private pointer.

We chose to have a thread-private pointer for the persistent state as well as for each transient state for several reasons. First, on some systems there is a limit to the size of thread local storage; therefore, all the state objects could not be made thread-local. Furthermore, to make the source code changes manageable, it is convenient to have a thread-local pointer instead of adding state arguments to each routine to pass around persistent and transient state objects.

The compiler uses a lot of global state either as file-scope static variables or external global objects. In our case, global variable references are generally direct. Our semi-automatic source transformation tool takes a compiler-generated listing of global variables defined in each module and from that creates structures for transient and persistent state objects. The tool also automatically redefines those global variables as macros with the proper implementations; that is, a dereference through a thread-local pointer to a field in a state object. This relieves us from the tedious and error-prone task of manually modifying all references to those variables.

By creating persistent and transient state objects as structures, we also help improve software engineering by organizing global data into logical objects that have well-defined lifetimes.

Of all the global data that needed to be privatized we only had to create synchronized access for about 3% of the global data. The breakdown of the classification of the remaining global data is illustrated in Figure 5.

Another major task in working with data-race detection tools is training them to understand customized memory pool operations that behave like `malloc` and `free`. It is common to allocate a memory block and use it in an iteration and free it in the same iteration. When a subsequent iteration allocates memory, it may get part or all of the freed block. If the data-race detection tool is not able to recognize the malloc-free pattern, it may report a large number of false dependence violations. The Intel Thread Checker recognizes a class of such operations. It also provides a mechanism through which the user can communicate this information with its runtime. This is achieved by means of an API call that passes a starting memory address and by the number of bytes to be considered as newly allocated memory chunks. In this way, the application architect asserts that the dependence relations across the specified barrier can safely be ignored. This is a simple mechanism; yet, it is capable of handling very complicated memory pool management systems.

PERFORMANCE RESULTS

After our compiler was successfully threaded and debugged, we spent some time in tuning its performance. Of particular importance was the choice of thread scheduling. We conducted many experiments with various parallel-loop scheduling policies. From the parallel-loop scheduling schemes supported by OpenMP*, self-scheduling provided the best performance. In addition, we implemented a scheduling policy that consistently outperformed self scheduling. The policy took advantage of the information that the compiler has about the functions it needs to compile. As part of parsing the input file and creating the intermediate language, the compiler has a substantial amount of information about the structure and the size of each function. We used this information as a static estimate of the time it would take to compile each function. We then grouped together functions in as many chunks as the number of threads or available cores in such a way that the workload of each chunk is almost the same. Through this technique we avoided the load imbalance problem. Figure 6 shows the parallel speedup we achieved in comparison to the theoretical speedup limit. The results are based on our experiments on a 4-socket dual-core system–a total of eight processors. We also spent some time in making sure lock contention was reduced by proper choice of locking. We were pleased with the final parallel performance of the threaded compiler as it approached the theoretical limit of parallel performance as dictated by Amdahl's law. Figure 6 shows the speedup of the threaded compiler compared to the original sequential

compiler when compiling the SPEC CPU2000 benchmarks.

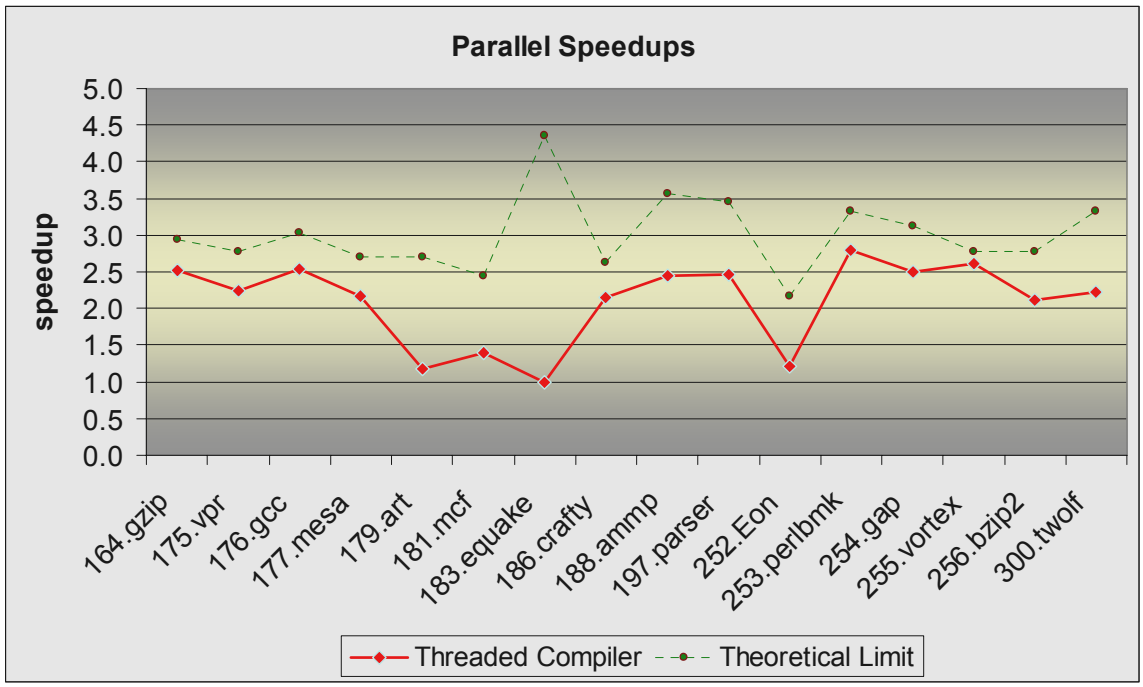


Figure 6: Parallel speedups of compiling CPU2000 benchmarks

CONCLUSION

We conclude that advancements in threading analysis tools have made parallelization of complex applications an easier task than what it was a decade or two ago. The overhead of the required instrumentation to perform the dynamic dependence checking has become affordable on modern microprocessors. Effective summarization and filtering of data dependence violations play a key role in managing the large problem size. We also found that semi-automatic mechanisms provide crucial help in accomplishing the repetitive and error-prone task of source code changes. Moreover, we found that good software engineering practices make threading easier.

ACKNOWLEDGEMENTS

We thank Zhiqiang Ma, Paul Petersen, and Victoria Gromova for their help with using and enhancing the Intel threading tools. Thanks also to Diana King, Sergey Kozhukhov, Suriya Madras-Subramanian, Xinmin Tian, and Ravi Ayyagari for their help with the static instrumentation of the Intel compiler. Throughout the entire project, we benefited from the mentorship of Kevin J. Smith.

REFERENCES

[1] Allen, R., and Kennedy, K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, San Francisco, CA, 2002.

[2] Banerjee, U., Bliss. B., Ma, Z., and Petersen, P., "Unraveling Data Race Detection in the Intel® Thread Checker," presented at the *First Workshop on Software Tools for Multi-core Systems (STMCS)*, in conjunction *with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, March 26, 2006, Manhattan, New York, NY.

[3] Intel® Compilers http://www3.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm

[4] Intel® Threading Analysis Tools http://www3.intel.com/cd/software/products/asmo-na/eng/threading/219785.htm

[5] Intel VTune™ Performance Analyzer http://www3.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

[6] Kuck, D.J., R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *8th annual ACM Symposium on*

Principles of Programming Languages, pp. 207–218, Jan. 26-28, 1981.

[7] Lamport, L., "Time, Clocks, and the Ordering of events in a Distributed System," *Communications of the ACM, Vol. 21, No. 7*, July 1978, pp. 558–565

[8] Lee, E. A., "The Problem with Threads," *IEEE Computer Society, Computer*, May 2006, Volume 39, Number 5.

AUTHORS' BIOGRAPHIES

Knud J. Kirkegaard is a Principal Engineer in the Intel Compiler Laboratory. He currently works on compiler optimizations for the Intel architectures. Since he joined Intel, he has worked on scalar optimizations, interprocedural optimizations, and profile guided optimizations for IA-32, Intel 64, and the Itanium processor family. His current interests are in optimized C++ code, compiler architecture, and thread-safe applications. He has an M.S. degree in Information and Control Systems Engineering from Aalborg University, Denmark. His e-mail is knud.j.kirkegaard at intel.com.

Mohammad Reza Haghighat is a Principal Engineer at Intel and the architect of Intel Compiler's code-coverage and test-prioritization tools. He is a threading expert and the author of *Symbolic Analysis for Parallelizing Compilers*, a book based on his pioneering Ph.D. research at the University of Illinois at Urbana-Champaign in the early 90s. Mohammad was the lead developer of one of the first Java JIT-Compilers and also has extensive experience in the performance aspects of database systems. More recently, he has been doing advanced development in the emerging Web 2.0 technologies such as AJAX and PHP. His e-mail is mohammad.r.haghighat at intel.com.

Ravi Narayanaswamy is a Senior Staff Engineer in the Intel Compiler Lab. He is currently working on software transaction memory support in the compiler. His previous role at Intel included porting of the compiler to various platforms. He was also involved in various optimizations in the compiler. He has an M.S. degree in Environmental Engineering and in Computer Science, both from Southern Illinois University, Carbondale. His e-mail is ravi.narayanaswamy at intel.com.

Bhanu Shankar is a Staff Engineer at Intel's Performance, Analysis and Threading Lab. His primary areas of interest include compilers, performance tools for HPC, and multi-threaded architectures. Bhanu received his Ph.D. degree from Colorado State University. His e-mail is bhanu.shankar at intel.com.

Neil Faiman is a Senior Staff Software Engineer in the Intel Compiler Lab, working on the development of tools to help assist users with threading their applications. Neil has been with Intel for five years. He came to Intel from Compaq, where he was the Intermediate Language architect for the GEM compiler project. Neil has B.S. and M.S. degrees from Michigan State University. His e-mail is neil.faiman at intel.com.

David Sehr heads the Advanced Tools team in the Software and Solutions Group at Intel. He was named a Senior Principal Engineer in 2003. At the time this work was done, David was the compiler architect and leader of the advanced development team in the Intel Compiler Lab. David received his Ph.D. degree from the University of Illinois at Urbana-Champaign in 1992, working under the direction of David Padua and Laxmikant Kale. His interests include threading, performance tools, language implementation and compilation, and static analysis. His e-mail is sehr at google.com.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm